Some good things to reference

Anonymous Records. You can read the above link for details, but the point of these is quite simple.

Records have been our main type for holding data for an observation. We've typically defined these ahead of time with a name before using them. This is good for important types that you will use frequently.

If you're using a particular record in only a few lines of code, then it can feel cumbersome to define the type beforehand. Anonymous records are a good solution in these circumstances. They are records that you can essentially use like regular records that we've been using, but you don't have to define the name of the record ahead of time.

I rarely use anonymous records, but you might find them useful for exploratory data manipulation. They're also kind of nice for these short problems because I don't need to define a record for each problem.

~~~
(*
# Anonymous records
1.
- Create a *record* named `ExampleRec` that has an `X` field of type int and a `Y` field of type int. Create an
example `ExampleRec` and assign it to a value named `r`.

- Create an *anonymous record* that has an `X` field of type int and a `Y` field of type int. Create an example of
the anonymous record and assign it to a value named `ar`.

2. Imagine you have this array

```fsharp
open System
type ArExample = { Date : DateTime; Value: float}
let arr = [|{ Date = DateTime(1990,1,1); Value = 1.25}
            { Date = DateTime(1990,1,2); Value = 2.25}
            { Date = DateTime(1991,1,1); Value = 3.25} |]
```
- Group the observations by a tuple of (year,month) and find the
minimum value for each group. Report the result as a tuple of the group
and the minimum value [so it will be ((year, month), minValue)].
- Now, the same thing with anonymous records.
Group the observations by an Anonymous Record {| Year = year; Month= month|} and find the
minimum value for each group. Report the result as an Anonymous record with a Group
field for the group and a value field for the minimum value [so it will be
{| Group = {| Year = year; Month= month|}; Value = minValue |}].

*)
~~~

~~~
(*
# Portfolio Returns

1. Imagine that you have the following positions in your portfolio.
For each position you have a weight and a return.
What is the return of the entire portfolio?

```fsharp
type PortReturnPos = { Id: string;  Weight: float; Return: float}
let stockPos = { Id = "stock"; Weight = 0.25; Return = 0.1 }
let bondPos = { Id = "bond"; Weight = 0.75; Return = 0.05}
```

2. Imagine that you have the following positions in your portfolio.
For each position you have a weight and a return.
What is the return of the entire portfolio?

```fsharp
type PortReturnPos = { Id: string;  Weight: float; Return: float}
let positions =
    [|{ Id = "stock"; Weight = 0.25; Return = 0.12 }
      { Id = "bond"; Weight = 0.25; Return = 0.22 }
      { Id = "real-estate"; Weight = 0.5; Return = -0.15 } |]
```

3. Imagine that you have the following positions in your portfolio.
For each position you have a weight and a return.
What is the return of the entire portfolio?

```fsharp
type PortReturnPos = { Id: string;  Weight: float; Return: float}
let positionsWithShort =
    [|{ Id = "stock"; Weight = 0.25; Return = 0.12 }
      { Id = "bond"; Weight = -0.25; Return = 0.22 }
      { Id = "real-estate"; Weight = 1.0; Return = -0.15 } |]
```

*)
~~~

~~~fsharp
(*
# Sharpe Ratios

1. Imagine that you have the following array of *annual* returns in
    excess of the risk-free rate. What is the *annualized* Sharpe ratio?

Note that the units are such that 0.1 is 10%.
```fsharp
#r "nuget: FSharp.Stats"
open FSharp.Stats
[| 0.1; -0.4; 0.2; 0.15; -0.03 |]
```

2. Imagine that you have the following array of *monthly* returns in
    excess of the risk-free rate. What is the *annualized* Sharpe ratio?

Note that the units are such that 0.1 is 10%.
```fsharp
#r "nuget: FSharp.Stats"
open FSharp.Stats
[| 0.1; -0.4; 0.2; 0.15; -0.03 |]
```

3. Imagine that you have the following array of *daily* returns in
    excess of the risk-free rate. What is the *annualized* Sharpe ratio?

Note that the units are such that 0.1 is 10%.
```fsharp
#r "nuget: FSharp.Stats"
open FSharp.Stats
[| 0.1; -0.4; 0.2; 0.15; -0.03 |]
```

*)
~~~

```fsharp
///********************
/// Answers
///********************
///
```

~~~fsharp
(*
# Anonymous records
1.
- Create a *record* named `ExampleRec` that has an `X` field of type int and a `Y` field of type int. Create an
example `ExampleRec` and assign it to a value named `r`.

- Create an *anonymous record* that has an `X` field of type int and a `Y` field of type int. Create an example of
the anonymous record and assign it to a value named `ar`.

2. Imagine you have this array

```fsharp
open System
type ArExample = { Date : DateTime; Value: float}
let arr = [|{ Date = DateTime(1990,1,1); Value = 1.25}
            { Date = DateTime(1990,1,2); Value = 2.25}
            { Date = DateTime(1991,1,1); Value = 3.25} |]
```
- Group the observations by a tuple of (year,month) and find the
minimum value for each group. Report the result as a tuple of the group
and the minimum value [so it will be ((year, month), minValue)].
- Now, the same thing with anonymous records.
Group the observations by an Anonymous Record {| Year = year; Month= month|} and find the
minimum value for each group. Report the result as an Anonymous record with a Group
field for the group and a value field for the minimum value [so it will be
{| Group = {| Year = year; Month= month|}; Value = minValue |}].

*)
~~~

```fsharp
// 1.
//

// a regular named record
type ExampleRec = { X : int; Y : int }
let r = { X = 1; Y = 2}
// an anonymous record. The difference is
// 1. We did not define the type ahead of time.
// 2. We put the pipe symbole "|" inside the curly braces.
let ar = {| X = 1; Y = 2|}
// Note that they are not the same type, so if you
// compare them they will be different even though
// the X and Y fields have the same values.
// For example, running `r = ar`
// will give a compiler error

// 2.
//
open System
type ArExample = { Date : DateTime; Value: float}
let arr = [|{ Date = DateTime(1990,1,1); Value = 1.25}
            { Date = DateTime(1990,1,2); Value = 2.25}
            { Date = DateTime(1991,1,1); Value = 3.25} |]

// here I will explicitly put year and month in the final result
arr
|> Array.groupBy(fun x -> x.Date.Year, x.Date.Month)
|> Array.map(fun (group, xs) ->
    let year, month = group // Explicitly access year, month; same as let a,b = (1,2)
    let minValue = xs |> Array.map(fun x -> x.Value)|> Array.min
    (year, month), minValue) // explicitly put it in the result

// here I will explicitly put year and month in the final result,
// but I will deconstruct them using pattern matching in the
// function input
arr
|> Array.groupBy(fun x -> x.Date.Year, x.Date.Month)
|> Array.map(fun ((year, month), xs) -> // Explicitly pattern match year, month in function input
    let minValue = xs |> Array.map(fun x -> x.Value)|> Array.min
    (year, month), minValue) // explicitly put it in the result

// or
// since I'm just returning the grouping variable, there's really
// no need to deconstruct it into year, month at any point.
arr
|> Array.groupBy(fun x -> x.Date.Year, x.Date.Month)
|> Array.map(fun (group, xs) -> // match group to (year,month) together
    let minValue = xs |> Array.map(fun x -> x.Value)|> Array.min
    group, minValue)

// Now using anonymous records
// This is where anonymous records can be useful.
// For example, sometimes grouping by many things,
// using anonymous records like this make it more clear what the different
// grouping variables are because they have names.
// It's like a middle ground between tuples with no clear naming structure
// and regular named records that are very explicit.
arr
|> Array.groupBy(fun x -> {| Year = x.Date.Year; Month = x.Date.Month |})
|> Array.map(fun (group, xs) ->
    let year, month = group.Year, group.Month // explicit deconstruct
    let minValue = xs |> Array.map(fun x -> x.Value)|> Array.min
    {| Group = {| Year = year; Month = month|}; Value = minValue |})

// or, do the same thing by returning the whole group without deconstructing
arr
|> Array.groupBy(fun x -> {| Year = x.Date.Year; Month = x.Date.Month |})
|> Array.map(fun (group, xs) ->
    let minValue = xs |> Array.map(fun x -> x.Value)|> Array.min
    {| Group = group; Value = minValue |})
```

~~~fsharp
(*
# Portfolio Returns

1. Imagine that you have the following positions in your portfolio.
For each position you have a weight and a return.
What is the return of the entire portfolio?

```fsharp
type PortReturnPos = { Id: string;  Weight: float; Return: float}
let stockPos = { Id = "stock"; Weight = 0.25; Return = 0.1 }
let bondPos = { Id = "bond"; Weight = 0.75; Return = 0.05}
```

2. Imagine that you have the following positions in your portfolio.
For each position you have a weight and a return.
What is the return of the entire portfolio?

```fsharp
type PortReturnPos = { Id: string;  Weight: float; Return: float}
let positions =
    [|{ Id = "stock"; Weight = 0.25; Return = 0.12 }
      { Id = "bond"; Weight = 0.25; Return = 0.22 }
      { Id = "real-estate"; Weight = 0.5; Return = -0.15 } |]
```

3. Imagine that you have the following positions in your portfolio.
For each position you have a weight and a return.
What is the return of the entire portfolio?

```fsharp
type PortReturnPos = { Id: string;  Weight: float; Return: float}
let positionsWithShort =
    [|{ Id = "stock"; Weight = 0.25; Return = 0.12 }
      { Id = "bond"; Weight = -0.25; Return = 0.22 }
      { Id = "real-estate"; Weight = 1.0; Return = -0.15 } |]
```

*)
~~~

```fsharp
// 1.
//
// Remember that portfolio returns are a weighted average
// of the returns of the stocks in the portfolio. The weights
// are the position weights.
type PortReturnPos = { Id: string;  Weight: float; Return: float}

let stockPos = { Id = "stock"; Weight = 0.25; Return = 0.1 }
let bondPos = { Id = "bond"; Weight = 0.75; Return = 0.05}

let stockAndBondPort =
    stockPos.Weight*stockPos.Return + bondPos.Weight*bondPos.Return
// or, doing the multiplication and summation with collections
let weightXreturn =
    [|stockPos;bondPos|]
    |> Array.map(fun pos -> pos.Weight*pos.Return)
// look at it
weightXreturn
// now sum
let stockAndBondPort2 = weightXreturn |> Array.sum
// check
stockAndBondPort = stockAndBondPort2 // evaluates to true

// 2.
//
let positions =
    [|{ Id = "stock"; Weight = 0.25; Return = 0.12 }
      { Id = "bond"; Weight = 0.25; Return = 0.22 }
      { Id = "real-estate"; Weight = 0.5; Return = -0.15 } |]
let threeAssetPortfolioReturn =
    positions
    |> Array.map(fun pos -> pos.Weight*pos.Return)
    |> Array.sum

// 3.
//
let positionsWithShort =
    [|{ Id = "stock"; Weight = 0.25; Return = 0.12 }
      { Id = "bond"; Weight = -0.25; Return = 0.22 }
      { Id = "real-estate"; Weight = 1.0; Return = -0.15 } |]
let positionsWithShortReturn =
    positionsWithShort
    |> Array.map(fun pos -> pos.Weight*pos.Return)
    |> Array.sum
```

~~~
(*
# Sharpe Ratios

1. Imagine that you have the following array of *annual* returns in
    excess of the risk-free rate. What is the *annualized* Sharpe ratio?

Note that the units are such that 0.1 is 10%.
```fsharp
#r "nuget: FSharp.Stats"
open FSharp.Stats
[| 0.1; -0.4; 0.2; 0.15; -0.03 |]
```

2. Imagine that you have the following array of *monthly* returns in
    excess of the risk-free rate. What is the *annualized* Sharpe ratio?

Note that the units are such that 0.1 is 10%.
```fsharp
#r "nuget: FSharp.Stats"
open FSharp.Stats
[| 0.1; -0.4; 0.2; 0.15; -0.03 |]
```

3. Imagine that you have the following array of *daily* returns in
    excess of the risk-free rate. What is the *annualized* Sharpe ratio?

Note that the units are such that 0.1 is 10%.
```fsharp
#r "nuget: FSharp.Stats"
open FSharp.Stats
[| 0.1; -0.4; 0.2; 0.15; -0.03 |]
```

*)
~~~

```fsharp
// 1.
//
#r "nuget: FSharp.Stats"
open FSharp.Stats

let rets = [| 0.1; -0.4; 0.2; 0.15; -0.03 |]
let retsAvg = rets |> Array.average
// we get stDev from FSharp.Stats
// there is only a Seq.stDev, not Array.stDev.
// We can use Seq.stDev with array because you
// can pipe any collection to a Seq.
let retsStdDev = rets |> Seq.stDev
let retsSharpeRatio = retsAvg/retsStdDev

// 2.
// remember that to annualize an arithmetic return,
// we do return * (# compounding periods per year)
// to annualize a standard deviation,
// we do sd * sqrt(# compounding periods per year)

let monthlyRetsAnnualizedAvg = 12.0*(rets |> Array.average)
// or
let monthlyRetsAnnualizedAvg2 =
    rets
    |> Array.average
    // now we're going to use a lambda expression.
    // this is the same idea as when we do Array.map(fun x -> ...)
    // except now we're only piping a float, not an array so
    // we're leaving off the "Array.map"
    |> (fun avg -> 12.0 * avg)
// or, in two steps
let monthlyRetsAvg = rets |> Array.average
let monthlyRetsAnnualizedAvg3 = 12.0*monthlyRetsAvg

// now the standard deviation
let monthlyRetsAnnualizedSd =
    rets
    |> Seq.stDev
    |> fun monthlySd -> sqrt(12.0) * monthlySd
//or, in two steps
let monthlyRetsSd = rets |> Seq.stDev
let monthlyRetsAnnualizedSd2 = sqrt(12.0)*monthlyRetsSd

// SharpeRatio
let annualizedSharpeFromMonthly =
    monthlyRetsAnnualizedAvg / monthlyRetsAnnualizedSd

// or, since 12.0/sqrt(12.0) = sqrt(12.0) then
// (monthlyRetsAvg *12.0)/(monthlyRetsSd*sqrt(12.0)) =
//      sqrt(12.0)*(monthlyRetsAvg/monthlyRetsSd)
let annualizedSharpeFromMonthly2 =
    sqrt(12.0) * (monthlyRetsAvg/monthlyRetsSd)

// check
// we have to round because floating point math gives us slightly different #'s
// recall from the fundamentals lecture how floating point math is inexact.
Math.Round(annualizedSharpeFromMonthly,6) = Math.Round(annualizedSharpeFromMonthly2,6) // true

// 3.
//
// Convention for daily is 252 trading days per year.
// so annualize daily by multiplying by sqrt(252.0)
let annualizedSharpeFromDaily =
    let avgRet = rets |> Array.average
    let stdevRet = rets |> Seq.stDev
    sqrt(252.0) * (avgRet/stdevRet)
// or in multiple steps
let dailyAvgRet = rets |> Array.average
let dailyStDevRet = rets |> Seq.stDev
let annualizedSharpeFromDaily2 =
    sqrt(252.0) * (dailyAvgRet/dailyStDevRet)


///********************
/// Questions
///********************
///
```